

Programmation Orientée Objet en Python

Classes et Objets

T. Dietenbeck

thomas.dietenbeck@sorbonne-universite.fr



- 1 Introduction
- 2 Classes
- 3 Objets
- 4 Exemple
- 5 Complément sur les classes
- 6 Exercices

1 Introduction

- Pourquoi des objets ?
- Qu'est-ce que l'UML ?
- Objectifs

2 Classes

3 Objets

4 Exemple

5 Complément sur les classes

6 Exercices

Les types

Les variables utilisées jusqu'à présent étaient :

- de types primitifs (**ex** : entier, réel, caractère, ...) ou des tableaux / chaînes de caractères
- accessibles et modifiables par tous
- sans lien explicite entre elles

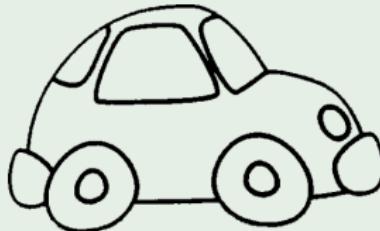
Les voitures

- Données

- une marque
- une année de fabrication

- Traitements

- Calculer son âge (année actuelle - année de fabrication)
- Afficher les informations relatives au véhicule (marque, année)



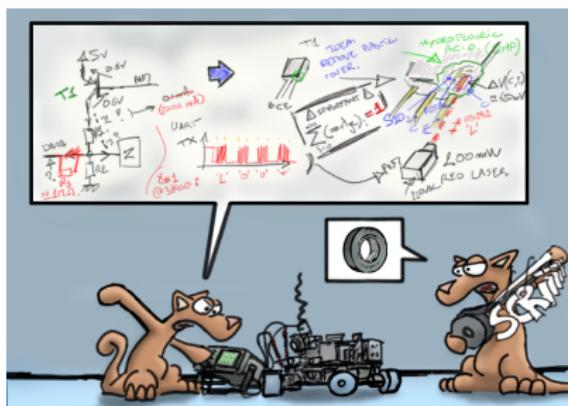
Code Python

```
def afficherVoiture( marque, annee ) :  
    ...  
def calculeAgeVoiture( annee ) :  
    ...  
  
marqueVoiture1 = "Renault"  
anneeVoiture1 = 2014  
afficherVoiture( marqueVoiture1, anneeVoiture1 )  
  
anneeVoiture1 = 42                      # Insense  
marqueVoiture1 = "Charlie"  
  
marqueVoiture2 = "Ferrari"  
anneeVoiture2 = 2012  
afficherVoiture( marqueVoiture1, anneeVoiture2 )  # KO  
  
age = calculeAgeVoiture( 1337 )
```



Limitations

- Tout ne peut pas être modélisé uniquement par des types primitifs
 - Certaines variables sont liées implicitement
 - ex : la marque et l'année de fabrication d'une voiture
 - Les modifications de certaines variables doivent être contrôlées
 - ex : l'année de fabrication d'une voiture
 - On ne peut appliquer certaines méthodes spécifiques qu'à des variables particulières
 - ex : le calcul de l'âge d'une voiture





Définition de l'UML selon l'OMG

Langage visuel dédié à la spécification, la construction et la documentation des artefacts d'un système logiciel

UML : Unified Modeling Language

- Langage de modélisation
 - Permet de représenter un système de manière abstraite
- Langage formel
 - Concis, précis et simple
- Langage graphique
 - Basé sur des diagrammes (13 pour UML 2.0)
- Standard industriel
 - Object Management Group (OMG)
 - Initié par Grady Booch, James Rumbaugh et Ivar Jacobson

[Alan Perlis]

- “Une fois que vous comprenez comment écrire un programme, trouvez quelqu'un d'autre pour l'écrire.”

UML, outil d'excellence pour

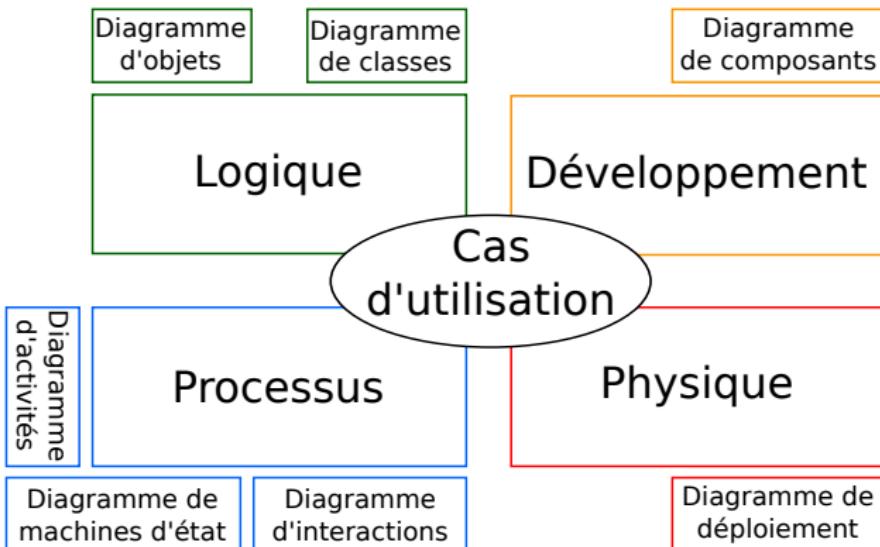
- Concevoir des logiciels informatiques
- Communiquer sur des processus logiciels ou d'entreprise
- Présenter, documenter un système à différents niveaux de détails

UML, généralisation aux domaines

- Secteur de la banque et de l'investissement
- Santé
- Défense
- Informatique distribuée
- Systèmes embarqués
- Secteur de la vente et de l'approvisionnement

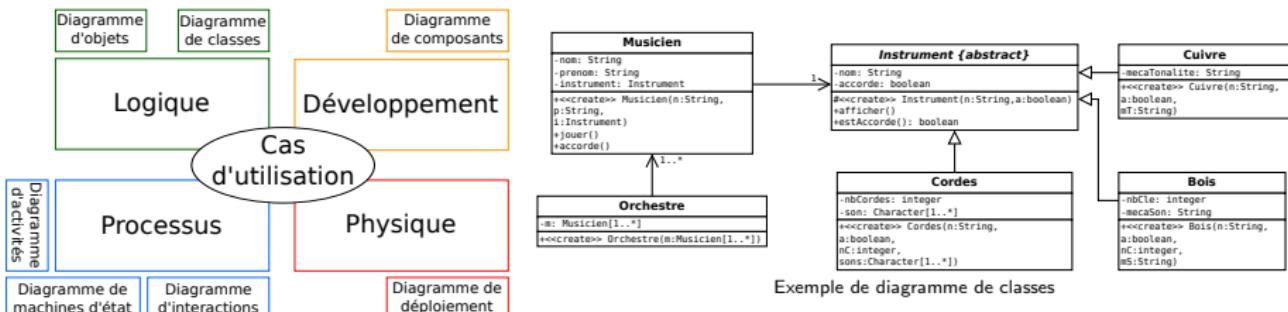
Modèle de vue

- Modèle de vue 4+1 (proposé par P. Kruchten)
- Organisation de l'analyse OO et des 13 diagrammes UML



Vue Logique

- Donne la description abstraite des parties d'un système
- Décrit les aspects statiques représentant la structure du problème
- Sert à modéliser les composants d'un système et leurs interactions (statiques)



Exemple de diagramme de classes

Objectifs

- Regrouper **plusieurs données** dans un **même type** (appelé classe)
- Regrouper **données** (\simeq les variables) et **traitements** (\simeq les fonctions)
- **Protéger** les informations (encapsulation)
- Diagrammes de classes : Décrire les classes d'une application et leurs relations statiques

Diagrammes de classes

- Très nombreuses utilisations
- Diagrammes fondamentaux : les plus connus et utilisés
- Permettent de modéliser plusieurs niveaux
 - conceptuel (domaine, analyse) (Modèle du domaine)
 - implémentation (code) (Modèle de conception)

1 Introduction

2 Classes

- Déclaration d'une classe
- Attributs et méthodes
- Constructeurs et destructeur
- Encapsulation et protection
- Diagramme de classes

3 Objets

4 Exemple

5 Complément sur les classes

6 Exercices

Une classe

Abstraction d'un concept

- regroupant :
 - **des données** (type primitif ou autre classe) propres à chaque instance.
 - **des traitements** (fonctions) communs à toutes les instances.
- permettant de :
 - **limiter l'accès** aux champs depuis l'extérieur de la classe
 - **contrôler les modifications** apportées aux champs

Vocabulaire

- Les données sont appelées les **attributs** de la classe.
- Les traitements sont appelés les **méthodes** de la classe.
- Attributs et méthodes sont appelés les **champs** de la classe.

Représentation UML

- Rectangle composé de compartiments
 - Compartiment 1 : Nom de la classe (commence par une majuscule, en gras)
 - Compartiment 2 : Attributs
 - Compartiment 3 : Méthodes
 - Possibilité d'ajouter des compartiments (exceptions, ...)
- Différents niveaux de détail possibles
 - Possibilité d'omettre attributs et/ou méthodes

MaClasse

MaClasse
-unAttribut: unType

MaClasse
-unAttribut: unType
+uneMethode()

Déclaration d'une classe en Python

Syntaxe

- Le mot clef `class`
- Le nom de la classe (par convention, la première lettre est en majuscule)
- Une classe est considérée comme un bloc. Sa déclaration se termine donc par un “`:`” et le corps de la classe est indenté.

Déclaration d'une classe

```
class MaClasse :  
    # Pas de declaration des attributs  
    # Declaration des methodes  
    uneMethode( self ) :  
        ...
```

MaClasse
-unAttribut: unType
+uneMethode()

Remarques

- En Python, les attributs d'une classe sont généralement déclarés dans le constructeur
- `self` permet de faire référence à soi-même

Représentation UML

accessibilité	nom :	type	[multiplicité]
+		integer,	facultatif
#		real,	(1 par défaut)
~		String,	
-		MaClasse,	...

Le nom et le type de l'attribut sont obligatoires

Remarque 1 : Les types

L'UML étant indépendant du langage de programmation, le type des attributs ne correspond pas forcément au type Python !

Ex : integer (UML) = int (Python), real (UML) = float (Python)

Représentation UML

accessibilité	nom :	type	[multiplicité]
+		integer,	facultatif
#		real,	(1 par défaut)
~		String,	
-		MaClasse, ...	

Le nom et le type de l'attribut sont obligatoires

Remarque 2 : La multiplicité

- Un attribut peut représenter un nombre quelconque d'objets de son type. 3 cas de figure :
 - ➊ On connaît le nombre d'éléments minimum et maximum :
 - nomAttribut :typeAttribut[valMin..valMax]
 - nomAttribut :typeAttribut[val] (cas où valMin = valMax)
 - ➋ On connaît le nombre d'éléments minimum :
 - nomAttribut :typeAttribut[valMin..*]
 - ➌ On ne connaît pas le nombre d'éléments :
 - nomAttribut :typeAttribut[*]

Représentation UML

accessibilité	nom :	type	[multiplicité]
+		integer,	facultatif
#		real,	(1 par défaut)
~		String,	
-		MaClasse,	...

Le nom et le type de l'attribut sont obligatoires

Remarque 2 : La multiplicité

- Un attribut peut représenter un nombre quelconque d'objets de son type.
- Une multiplicité > 1 indique l'utilisation d'un tableau :
 - ➊ Tableau 1D : `typeAttribut[valMin..valMax]` ou `typeAttribut[0..*]`
 - ➋ Tableau 2D (Matrice) : `typeAttribut[lMin..lMax, cMin..cMax]` ou `typeAttribut[0..*, 0..*]`

Représentation UML

accessibilité	nom(paramètres) :	type de retour
+		nom : type		facultatif
#		séparé par		
-		une virgule		
	~			

Le nom de la méthode et les parenthèses (même vides) sont obligatoires

Remarques

- Le type de retour ou d'un paramètre suit les mêmes règles que pour les attributs
- Si une méthode n'a pas de type de retour, elle ne renvoie rien

Définition en Python

- Déclaration classique de fonction
- Par convention le nom de la méthode commence par un verbe en minuscule
- Toute méthode a au moins un paramètre qui s'appelle toujours `self` (premier paramètre dans les parenthèses)

Remarques

- La déclaration d'une méthode suit les mêmes règles et principes que la déclaration de fonction

Représentation UML de méthodes

`-calculerAire() : real` \Leftrightarrow une méthode **calculerAire** de visibilité **privée**, sans paramètres et renvoyant un réel.

`+afficherSommet(x : integer)` \Leftrightarrow une méthode **afficherSommet** de visibilité **publique**, ayant un paramètre d'entrée **x** de type **entier** et ne renvoyant aucun résultat (pas de `return` dans son implémentation Python).

Déclaration Python de méthodes

```
# Methode privee calculerAire sans parametres
def __calculerAire( self ) :
    ...
    return aire      # un reel contenant l'aire

# Methode publique afficherSommet ayant un parametre d'entree x
def afficherSommet( self, x ) :
    ...
```

Constructeur et destructeur

- Mécanisme générique et systématique d'initialisation et de suppression d'un objet
- Le constructeur est appelé à chaque instantiation / création d'un nouvel objet
- Le destructeur est appelé à chaque suppression d'un objet

Remarques

- En Python, la destruction est gérée automatiquement par le Garbage Collector (destructeur implicite)
- On peut cependant définir explicitement un destructeur (méthode `__del__`)

Représentation UML d'un constructeur

Pour représenter un constructeur,

- on ajoute un stéréotype entre la visibilité et le nom de la méthode
- on donne le nom de la classe au constructeur

Syntaxe : + <<create>> NomClasse(paramètres éventuels)

Déclaration d'un constructeur en Python

- Comme une autre méthode mais avec `__init__` comme nom
- **Rôle :** Initialiser **tous les attributs** en leur affectant
 - soit des valeurs passées en paramètres
 - soit des valeurs par défaut (e.g. aujourd'hui pour une date, liste vide, ...)
- La méthode `__init__` ne renvoie **jamais** de résultat

Représentation UML d'un constructeur

+ <<create>> Vehicule() ⇔ constructeur sans paramètre de la classe Vehicule
+ <<create>> Personne(nom : String, prenom : String) ⇔ constructeur de la classe Personne avec 2 chaînes de caractères comme paramètres

Déclaration d'un constructeur en Java

```
class Vehicule :  
    # Constructeur sans parametre de la classe Vehicule  
    def __init__( self ) :  
        ...  
  
class Personne :  
    # Constructeur de la classe Personne avec 2 parametres  
    def __init__( self, nom, prenom ) :  
        ...
```

La classe Voiture

- 2 attributs : année et marque
- 2 méthodes : afficher et calculerAge

```
import datetime
class Voiture :
    def __init__( self , année , marque ) :
        self.année = année
        self.marque = marque

    def afficher( self ) :
        print( "Voiture de marque " , self.marque , " fabrique en " ,
              self.année )

    def calculerAge( self ) :
        now = datetime.datetime.now()
        return now.year - self.année
```

Voiture
-marque: String
-année: integer
+<<create>> Voiture(année:integer, marque:String)
+afficher()
+calculerAge(): integer

Remarque

- Depuis une méthode, l'accès aux attributs de la classe ou l'appel d'autres méthodes de la classe se fait à l'aide de `self`.

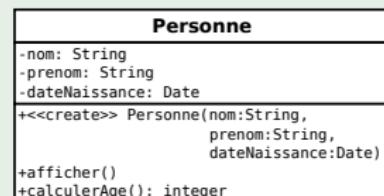
La classe Personne

- 3 attributs : nom, prenom et dateNaissance
- 2 méthodes : afficher et calculerAge

```
import datetime
class Personne :
    def __init__( self , nom , prenom , dateNaissance ) :
        self.nom = nom
        self.prenom = prenom
        self.dateNaissance = dateNaissance

    def afficher( self ) :
        print( self.prenom , " " , self.nom )

    def calculerAge( self ) :
        now = datetime.datetime.now()
        return now.year - self.dateNaissance.year
```



Remarque

- Depuis une méthode, l'accès aux attributs de la classe ou l'appel d'autres méthodes de la classe se fait à l'aide de **self**.

Protection des données et des traitements

Principe de l'encapsulation

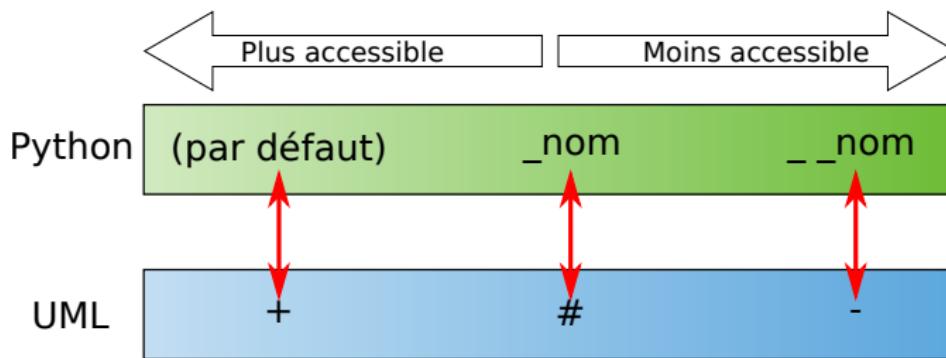
Masquer au maximum les champs de la classe pour restreindre leur accès depuis l'extérieur

Objectif

Protéger et contrôler les valeurs des attributs

⇒ **Cacher certaines données et certains traitements** (encapsulation) pour éviter que des méthodes extérieures à la classe puisse les modifier / utiliser

Protection des données et des traitements



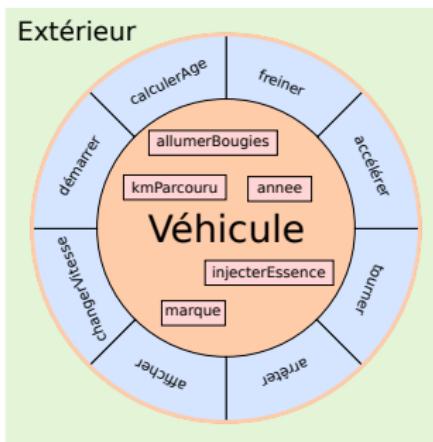
Modificateurs d'accès

Nom	UML	Python	Accessibilité
Public	+	Aucun (par défaut)	Pas de restriction. N'importe quelle classe
Protégé	#	<code>_</code> avant le nom	Classes dérivées
Privé	-	<code>__</code> avant le nom	Seules les méthodes situées dans la classe

Protection des données et des traitements

Accessibilité

- On peut définir l'**accessibilité** à chaque composante de la classe à l'aide de **spécificateurs**
- Le choix de l'accessibilité dépend de la nature des attributs et des méthodes
 - `démarrer` → + (public)
 - `allumerBougies, injecterEssence` → - (privé)
- Méthodes publiques = interface fonctionnelle / partie visible de l'objet : responsables de l'accès aux autres composantes



Principe de l'encapsulation (en Python)

- Si l'accès (en lecture ou en écriture) à l'attribut doit être contrôlé ou impossible ⇒ attribut privé :
 - la position du levier de vitesse d'une voiture doit être compris entre 1 et 6 (plus la marche arrière)
 - la note d'un étudiant doit être comprise entre 0 et 20
 - le mot de passe pour se connecter à son compte en banque
- Si l'implémentation interne de l'attribut ne correspond pas à sa représentation ⇒ attribut privé :
 - l'année de fabrication est stockée comme la différence avec l'année 1900
 - la pression est stockée en mmHg mais affichée en Pa
- Dans tous les autres cas ⇒ attribut public

Remarque

- L'objet lui-même (*i.e.* les méthodes qu'il contient) a **toujours** un accès direct à ses attributs (même s'ils sont privés).

Getters, Setters et Properties

- Intérêts :

- accès **contrôlé** aux attributs privés depuis l'extérieur
- ne pas affecter les autres classes si l'implémentation interne change (ex : au lieu de stocker l'année de création du Véhicule, on garde la différence entre 1900 et sa date de création)
- ajout d'appels à d'autres fonctions nécessaires après une lecture / écriture (ex : calcul de la représentation polaire d'un nombre complexe après la mise à jour de sa partie réelle)
- Leurs implémentation n'est pas obligatoire (ex : pas d'accès en lecture / écriture à un mot de passe)

Getters, Setters et Properties

- Accès en lecture
- Aucun paramètre d'entrée (autre que `self`)
- La ligne précédant la méthode est "décorée" : `@property`

Setters

- Accès **contrôlé** en écriture
- Un paramètre d'entrée (en plus de `self`)
- La ligne précédant la méthode est "décorée" : `@nomAttribut.setter`

Remarques

- En Python, les getters et setters sont appelés `property`
- Par convention, leur nom est celui de l'attribut

La classe Vehicule

```
class Vehicule :  
    def __init__( self, annee, marque ) :  
        # Implementation interne: difference avec 1900  
        if (annee > 1900) and (annee <= now.year)  
            self.__annee = annee - 1900      # => attribut prive  
        # Pas d'accès contrôlé => attribut public  
        self.marque = marque  
  
    @property  
    def annee( self ) :  
        return self.__annee + 1900  
    @annee.setter  
    def annee( self, annee ) :  
        # On vérifie qu'annee peut correspondre à une année de  
        if (annee > 1900) and (annee <= now.year)  # construction  
            self.__annee = annee - 1900
```

Remarque : le constructeur fait la même opération sur annee que le setter \Rightarrow solution pas très "pythonique"

La classe Vehicule : solution "pythonique"

```
class Vehicule :  
    def __init__( self , annee , marque ) :  
        # Implementation interne: difference avec 1900  
        self.annee = annee # Attribut prive avec appel du setter  
        # Pas d'accès contrôle => attribut public  
        self.marque = marque  
  
    @property  
    def annee( self ) :  
        return self.__annee + 1900  
    @annee.setter  
    def annee( self , annee ) :  
        # On vérifie qu'annee peut correspondre à une année de  
        if (annee > 1900) and (annee <= now.year) # construction  
            self.__annee = annee - 1900  
  
    # Pas de property pour marque car public
```

Avantage : aucune modification du constructeur si la visibilité de l'attribut annee est rechangée en public

La classe Velo

```
class Velo :  
    ...  
    @property  
    def numVitesse( self ) :  
        return self.__numVitesse  
    @numVitesse.setter  
    def numVitesse( self , num ) :  
        if (num > 0) and (num < 7) :  
            self.__numVitesse = num
```

La classe Date

```
class Date :  
    ...  
    @property  
    def jour( self ) :  
        return self.__jour  
    @jour.setter  
    def jour( self , jour ) :  
        if (jour > 0) and (jour < 31) :  
            self.__jour = jour  
            return True      # Modification réussie  
        return False     # Modification ratée
```

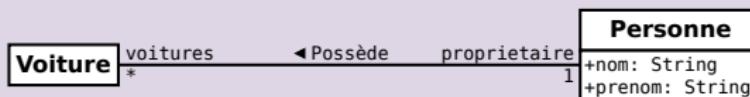
Objectifs

- Décrire les classes d'une application et leurs relations statiques

Principe

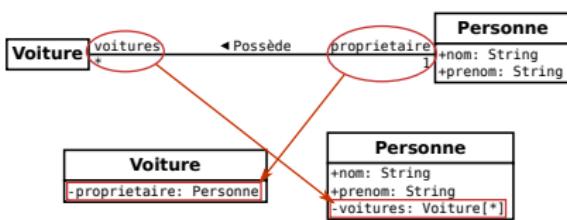
- Les associations sont utilisées pour matérialiser les relations entre éléments qui font partie de la modélisation.
- 2 manières de modéliser une relation :

- “en ligne”



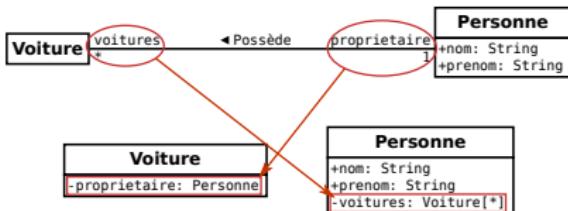
- par “association”





Problème

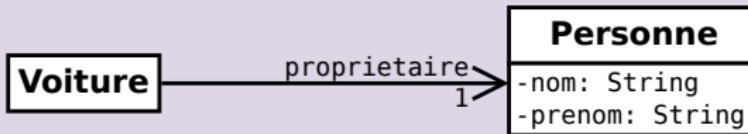
Il y a un attribut (e.g. `proprietaire` ou `voitures`) dans chaque classe (e.g. `Voiture` ou `Personne`)
⇒ Comment rendre l'association unidirectionnelle ?



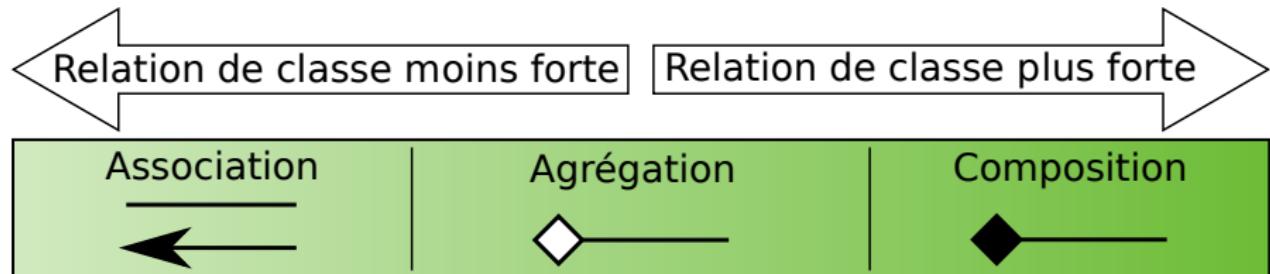
Problème

Il y a un attribut (e.g. `proprietaire` ou `voitures`) dans chaque classe (e.g. `Voiture` ou `Personne`)
 ⇒ Comment rendre l'association unidirectionnelle ?

Navigabilité



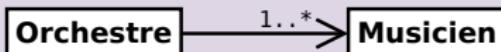
- Voiture contient un attribut `proprietaire` de type Personne
- Personne ne contient pas d'attribut de type Voiture



Les relations

- Association : une classe contient une référence à un (ou plusieurs) objet(s) d'une autre classe
- Agrégation : une classe détient et peut partager un (ou plusieurs) objet(s) d'une autre classe
- Composition : une classe détient et ne partage pas un (ou plusieurs) objet(s) d'une autre classe

Les relations

- Association : 

```
graph LR; Orchestre[Orchestre] -- "1..*" --> Musicien[Musicien]
```

 - une classe contient une référence à un (ou plusieurs) objet(s) d'une autre classe
 - une classe fonctionne avec un objet d'une autre classe
- Agrégation : 

```
graph LR; Musicien[Musicien] -- "1" --> Instrument[Instrument]
```

 - une classe détient et peut partager un (ou plusieurs) objet(s) d'une autre classe
 - une classe est propriétaire d'un objet d'une autre classe (modification, création possible)
- Composition : 

```
graph LR; Violon[Violon] -- "1..*" --> Corde[Corde]
```

 - une classe détient et gère un (ou plusieurs) objet(s) d'une autre classe : création et destruction
 - une classe ne partage pas l'objet avec d'autre classe (*i.e.* multiplicité de 1 d'un côté de la relation)

Quelques questions à se poser

- Indépendance des objets (\Rightarrow association) ou assymétrie et lien de subordination entre instances des deux classes (\Rightarrow agrégation ou composition) ?
- Propagation d'attributs ou d'opérations du tout vers les parties ? \Rightarrow agrégation ou composition
- Création et destruction des parties avec le tout ? \Rightarrow composition

Remarques

- Dans le doute, toujours utiliser une association (c'est la moins contrainte)
- Le choix entre composition et agrégation peut être laissé à la phase de conception

1 Introduction

2 Classes

3 Objets

- Qu'est ce qu'un objet
- Initialisation et destruction
- Accès aux composantes

4 Exemple

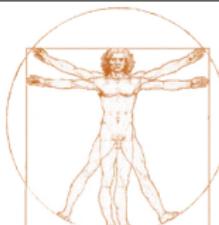
5 Complément sur les classes

6 Exercices

Définition

- Réalisation concrète d'une description abstraite
- Objet (comparable à une variable) = Instance d'une classe (comparable à un type)
- Un objet (*e.g.* une voiture particulière) est créé (instancié) à partir d'une classe (*e.g.* sa classe Voiture).
- Chaque classe détient le "plan de construction" (*i.e.* le constructeur) pour chacun de ses objets.

Différence Classe / Objet

Classe		Objet	
puissance			5cv
annee			2015
kmParcouru			30000km
nom			Norris
prenom			Chuck
dateNaissance			10/03/1940
taille			1.78m
race			Naine d'Asie
sexe			Femelle
age			6 ans

Durée de vie d'un objet

- ➊ Création : Appel à un constructeur

```
var = NomClasse( parametres )
```

- ➋ Utilisation : Appel de méthode, lecture / écriture des attributs, ...

- ➌ Destruction : Appel à un destructeur

- Etape facultative (appel implicite du destructeur)
- Appel explicite du destructeur : `del var`

Création / Instanciation d'objets

Création / Instanciation d'objets

Pour créer (ou instancier) un objet, il faut :

- Stocker dans une variable l'objet créé par appel du constructeur : la variable pointe sur le nouvel objet

Exemples d'instanciation

```
v1 = Vehicule() # Creation d'un objet de type Vehicule
# Creation d'un objet de type Personne (et d'une Date)
p1 = Personne( "Norris", "Chuck", Date(10, 03, 1940) )
```

Déclaration de la classe

```
class Voiture :  
    def __init__( self , annee , marque ) :  
        self.annee = annee  
        self.marque = marque  
  
    ...
```

Appel du constructeur (dans un programme principal par ex.)

```
v1 = Voiture( 2014 , "Renault" )
```

Destructeur implicite : le garbage collector

Principe

- En Python, lorsqu'il n'existe plus aucune référence sur un objet, on est certain que le programme ne pourra plus y accéder
 - ⇒ L'objet devient candidat au ramasse-miettes, l'espace mémoire sera libéré (mais pas toujours immédiatement)

On peut forcer la récupération de la mémoire en appelant explicitement le destructeur d'un objet

```
v = new Voiture( 2014, "Renault")
# Code qui utilise v
...
# Destruction de v
del v
# Code encore long
```

Accès aux composantes

- L'accès aux composantes de l'objet (attributs et méthodes) est réalisé par l'opérateur “.”
- Syntaxe : `nomObjet . nomChamp`

Exemple d'accès aux composantes

```
v1 = Vehicule()
v1.marque = "Renault"      # En supposant les attributs publics
v1.annee = 2014            # ou que les setters existent
v1.afficher()              # Affiche: "Vehicule de marque Renault fabrique
                           ↪ en 2014"
```

1 Introduction

2 Classes

3 Objets

4 Exemple

- Déclaration de la classe Véhicule
- Utilisation de la classe Véhicule
- Erreurs courantes

5 Complément sur les classes

6 Exercices

Énoncé

- Données

- une marque
 - une année de fabrication

- Traitements

- Calculer son âge (année actuelle - année de fabrication)
 - Afficher les informations relatives au véhicule (marque, année)

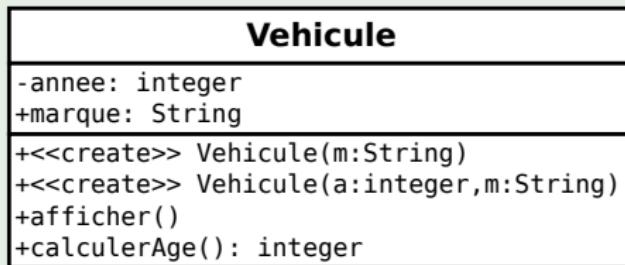
- Contraintes

- L'année de fabrication devra être comprise entre l'année actuelle et 1900

Analyse

- Une classe Vehicule
- 2 attributs
 - marque : chaîne de caractères, public
 - année : entier, privé
- 2 méthodes (en plus du constructeur)
 - calculerAge : pas de paramètres d'entrée, renvoie un entier, public
 - afficher : pas de paramètres d'entrée, pas de valeur de retour, public

Diagramme UML de la classe Vehicule



Déclaration Python de la classe Vehicule

```
# Import des packages nécessaires au fonctionnement
from datetime import datetime # de la classe

class Vehicule :
    # 1) Déclaration des constructeurs
    def __init__( self, marque, annee = datetime.now( ).year ) :
        self.marque = marque
        self.annee = annee # Année actuelle par défaut

    # 2) Déclaration des property
    @property
    def annee( self ) :
        return self.__annee
    @annee.setter
    def annee( self, annee ) :
        anneeNow = datetime.now( ).year
        if (annee > 1900) and (annee <= anneeNow) :
            self.__annee = annee
```

Déclaration Python de la classe Vehicule

```
# 3) Déclaration des méthodes
def afficher( self ) :
    res = "Véhicule"
    res += " de marque " + self.marque
    res += " fabrique en " + str( self.annee )

    print( res )

def calculerAge( self ) :
    anneeNow = datetime.now( ).year
    return anneeNow - self.annee
```

Utilisation de la classe Vehicule

```
# Creation d'un Vehicule
v1 = Vehicule( "Peugeot", 2014 )
# Affichage de ses informations
v1.afficher( )
# Affichage de son age
print( "v1 a", v1.calculerAge( ), "ans." )

# Creation d'un autre Vehicule
v2 = Vehicule( "Renault" )
# Modification de son annee de fabrication
v2.annee = 2012
```

Erreurs courantes

```
v1.afficher( ) # Utilisation d'un objet non cree

v2 = Vehicule( "Renault", 2012 )
afficher( v2 )      # Mauvaise syntaxe d'appel

del v2
v2.afficher( )      # Appel d'une methode sur un objet detruit

v3 = Vehicule( )   # Mauvais appel du constructeur
```



1 Introduction

2 Classes

3 Objets

4 Exemple

5 Complément sur les classes

- Constructeurs multiples
- Égalité d'objets
- Affichage d'objets
- Agrégation ou composition ?

6 Exercices

Constructeurs multiples

- En Python, il est impossible de définir plusieurs méthodes ayant le même nom
- Comment gérer le cas où plusieurs constructeurs pourraient exister pour une même classe ?

Exemples

- Pour une Voiture, si l'année de fabrication n'est pas donnée, on considérera l'année courante
- Une Grille peut être créée soit par un tableau de chaînes de caractères, soit de manière aléatoire à partir de ses dimensions

Constructeurs multiples

- En Python, il est impossible de définir plusieurs méthodes ayant le même nom
- Comment gérer le cas où plusieurs constructeurs pourraient exister pour une même classe ?

Exemples

- Pour une Voiture, si l'année de fabrication n'est pas donnée, on considérera l'année courante
- Une Grille peut être créée soit par un tableau de chaînes de caractères, soit de manière aléatoire à partir de ses dimensions

Solutions

- Utiliser des valeurs par défaut (*cf.* déclaration de fonctions)
- Utiliser des méthodes de classe

Constructeurs multiples et méthodes de classe

- Une méthode de classe est une méthode qui peut s'appeler soit depuis une classe soit d'un objet (déconseillé) :
 - elle est précédée du décorateur `@classmethod`
 - le premier paramètre est `cls` au lieu de `self`
 - dans le cas d'un constructeur, elle fait appel au constructeur de la classe dont elle renvoie le résultat
- comme elle s'applique à la classe, elle n'a pas accès aux attributs d'un objet !

Exemple

```
from datetime import datetime
class Vehicule :
    # Constructeur "normal": tous les attributs sont connus
    def __init__( self, annee, marque ) :
        self.annee = annee
        self.marque = marque

    # Methode / Constructeur de classe: gestion de l'annee
    @classmethod
    def fromMarqueOnly( cls, marque ) :
        annee = datetime.now( ).year    # Annee actuelle par defaut
        return cls( annee, marque )    # Appel du constructeur de
                                      ↪ Vehicule

    ... # Reste de la classe

v1 = Vehicule.fromMarqueOnly( "Renault" )
v1.afficher( ) # Vehicule de marque Renault de 2019
```

Égalité d'objets

- Par défaut, l'opérateur `==` compare les adresses des objets et non leurs valeurs / attributs.

```
v1 = Vehicule( "Peugeot", 2014 )
v2 = Vehicule( "Peugeot", 2014 )

print( v1 == v2 )  # Affiche "False"
```

La méthode equals

- Tout objet possède une méthode `__eq__` qui permet de vérifier si 2 objets sont identiques :
Syntaxe : `def __eq__(self, other)`
- Le test d'égalité passe par les étapes suivantes :
 - ① On s'assure que l'autre objet existe (*i.e.* il n'est pas `None`) et de même type que l'objet courant (mot clé `isinstance`)
 - ② On compare les adresses de stockage (s'ils pointent vers la même adresse, ce sont les mêmes objets)
 - ③ On compare les attributs qui font l'unicité d'un objet (*e.g.* numéro d'étudiant, code d'une UE, parties réelle et imaginaire d'un complexe, ...).

Remarque

- Si des attributs comparés sont des classes, il faut aussi implémenter la méthode `__eq__` dans ces classes
- Lorsque la méthode `__eq__` est implementée pour une classe, elle est appelée par `==`

Exemple d'implémentation de la méthode `__eq__` : Cas général

```
def __eq__( self, other ) :
    # 1/ On commence par verifier que other est un MaClasse
    # Rq: isinstance verifie aussi que other != None
    if not isinstance( other, MaClasse ) :
        return False
    # 2/ Meme adresse => meme objet
    if( self is other ) :
        return True

    # Si on arrive ici, c'est qu'on a un objet de type MaClasse
    #   ↪ stocke ailleurs en memoire
    # 3/ On compare les champs d'interet
    #   Egalite d'attribut
    if self.att1 != other.att1 :
        return False
    if self.att2 != other.att2 :
        return False
    ...  #Tous les attributs d'interet

    # Tous les attributs sont les memes => meme objet
    return True
```

Exemple d'implémentation de la méthode `__eq__` : Classe Vehicule

```
def __eq__( self, other ) :
    # 1/ On commence par vérifier que o est un Vehicule
    if not isinstance( other, Vehicule ) :      return False
    # 2/ Meme adresse => meme objet
    if self is other :                         return True
    # On a un objet de type Vehicule
    # 3/ On compare les champs d'intérêt (marque et année)
    if self.marque != other.marque :
        return False
    if self.annee != other.annee :
        return False
    # Tous les attributs sont les mêmes => meme objet
    return True
```

Exemple d'appel de la méthode `__eq__` : Classe Vehicule

```
v1 = Vehicule( "Peugeot", 2014 )
v2 = Vehicule( "Peugeot", 2014 )

print( v1 == v2 )  # Affiche "True"
```

Affichage d'objets

```
v1 = Vehicule( 2014, "Peugeot" )
print( v1 ) # "<__main__.Vehicule object at 0x7fa3cf95c400>"
```

- L'affichage d'un objet par la fonction `print` donne sa classe suivie de l'adresse où il est stocké
- Comment afficher la marque et l'année de fabrication d'un véhicule sans passer par notre méthode `afficher` ?

Affichage d'objets

```
v1 = Vehicule( 2014, "Peugeot" )
print( v1 ) # "<__main__.Vehicule object at 0x7fa3cf95c400>"
```

- L'affichage d'un objet par la fonction print donne sa classe suivie de l'adresse où il est stocké
- Comment afficher la marque et l'année de fabrication d'un véhicule sans passer par notre méthode afficher ?

La méthode __str__

- Tout objet possède une méthode __str__ qui permet de convertir l'objet en chaîne de caractères :
Syntaxe : `def __str__(self)`
- Si des attributs à convertir sont des classes, il faut aussi implémenter la méthode __str__ dans ces classes
- En UML, on représentera cette méthode par `+toString() : String`

Exemple d'implémentation de la méthode `__str__`

```
# Dans la classe Vehicule
def __str__( self ) :
    res = "Vehicule"
    res += " de marque " + self.marque
    res += " fabrique en " + str( self.annee )

    return res
```

Exemple d'appel de la méthode `__str__`

```
# Dans le programme principal
v1 = Vehicule( 2014, "Peugeot" )
print( v1 ) # Appel implicite a __str__
# Affiche "Vehicule de marque Peugeot fabrique en 2014"
```

Exemple d'implémentation de la méthode `__str__`

```
# Dans la classe Vehicule
def __str__( self ) :
    res = "Vehicule"
    res += " de marque " + self.marque
    res += " fabrique en " + str( self.annee )

    return res
```

Intérêt de la méthode `__str__` (comparativement à afficher)

```
# Dans le programme principal
v1 = Vehicule( 2014, "Peugeot" )
print( v1, "qui a", v1.calculerAge( ), "ans" )
# "Vehicule de marque Peugeot fabrique en 2014 qui a 5 ans"

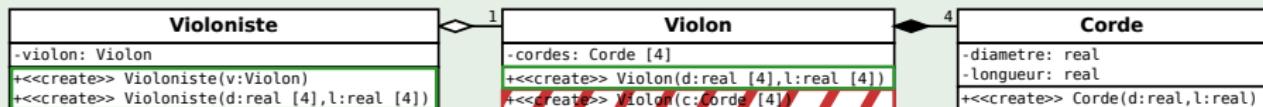
v1.afficher( )
print( "qui a", v1.calculerAge( ), "ans" )
# "Vehicule de marque Peugeot fabrique en 2014"
# "qui a 5 ans"
```

Différence entre agrégation et composition

Constructeur

- Agrégation : le constructeur peut soit prendre un objet déjà créé (auquel cas il est partagé) ou le créer lui même (par un appel au constructeur, partage ultérieur possible).
- Composition : pour empêcher l'attribut d'être partagé, le constructeur créera toujours l'attribut par un appel au constructeur.

Exemple - Agrégation



```

class Violoniste :
    def __init__( self , violon ) :
        self.violon = violon # le violon est deja cree (partage?)

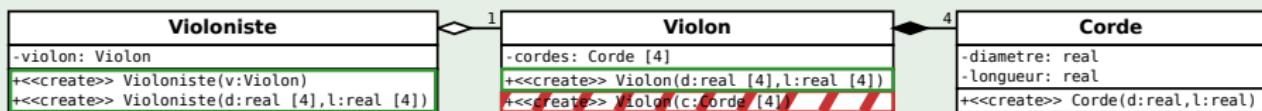
    @classmethod
    def fromViolon( cls , d , l ) :
        violon = Violon( d , l ) # un nouveau violon est cree
        return cls( violon ) # appel du constructeur avec le violon
  
```

Différence entre agrégation et composition

Constructeur

- Agrégation : le constructeur peut soit prendre un objet déjà créé (auquel cas il est partagé) ou le créer lui même (par un appel au constructeur, partage ultérieur possible).
- Composition : pour empêcher l'attribut d'être partagé, le constructeur créera toujours l'attribut par un appel au constructeur.

Exemple - Composition



```

class Violon
  def __init__( self, d, l ) :
    # Appel du constructeur de l'attribut
    self.cordes = [ Corde( d[i], l[i] ) for i in range(4) ]

  # Pas de constructeur __init__( c ) car pourrait entraîner un
  # → partage des cordes
  
```

Différence entre agrégation et composition

Constructeur

- Agrégation : le constructeur peut soit prendre un objet déjà créé (auquel cas il est partagé) ou le créer lui même (par un appel au constructeur, partage ultérieur possible).
- Composition : pour empêcher l'attribut d'être partagé, le constructeur créera toujours l'attribut par un appel au constructeur.

Getter & Setter

- **Avis personnel** : Particularité uniquement pour la composition.
- pour empêcher l'attribut d'être partagé, on ne créera ni getters ni setters

- 1 Introduction
- 2 Classes
- 3 Objets
- 4 Exemple
- 5 Complément sur les classes
- 6 Exercices

Énoncé

On souhaite modéliser un championnat de Basketball. Un championnat est composé de plusieurs équipes qui se rencontrent toutes 2 fois (matchs aller-retour). Des points sont attribués à chaque équipe en fonction du résultat de chaque match et ces points servent à produire le classement final du championnat.

Pour chaque match, on conservera les 2 équipes ayant jouées, la date de la rencontre ainsi que le score final. Le calendrier des matchs étant fixé en début de saison, on prévoira un moyen de faire jouer le match à la date prévue ainsi que de le reporter à une autre date (en cas de conflit avec d'autres compétitions sportives par exemple). En cas de report, on indiquera s'il est possible ou pas (en fonction du calendrier des 2 équipes).

Une équipe de Basket est composée d'au minimum 5 joueurs qui seront représentés par leur nom, poste et numéro. On conservera aussi l'équipe dans laquelle joue une personne. Un joueur peut cependant être sans contrat et ne pas avoir de club. En cours de saison, une équipe peut acheter de nouveaux joueurs ou en transférer vers un autre club.

Question

- Donner le diagramme UML de cette application

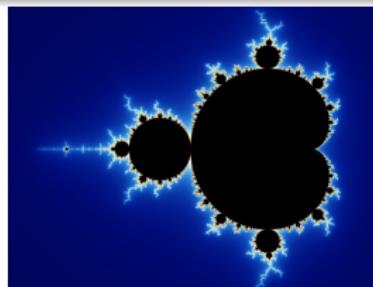
Énoncé

L'ensemble de Mandelbrot est une fractale définie comme l'ensemble des points c du plan complexe pour lesquels la suite de nombres complexes définie par récurrence par :

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$
 est bornée. En pratique, on peut montrer que si $|z_n| > 2$, alors la suite diverge et c n'appartient pas à l'ensemble de Mandelbrot.

Question

- Donner le diagramme UML de la classe Complexe
- Donner son implémentation
- Déterminer la valeur de n à partir de laquelle $|z_n| > 2$. On s'arrêtera au bout de m itérations si la suite n'a pas divergée.

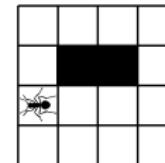
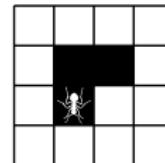
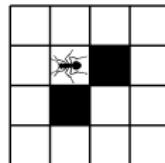
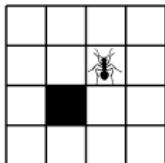


Énoncé

La fourmi de Langton est un automate cellulaire reposant sur des règles très simples :

- On considère une grille de $W \times H$ cases dont chacune peut être noire '#' ou blanche '.'.
- On dispose la fourmi aux coordonnées (x, y) de la grille
- On oriente la fourmi vers l'un des quatre points cardinaux (N : haut, E : droite, S : bas, W : gauche)
- On laisse la fourmi se déplacer pendant T tours de la façon suivante :
 - elle inverse la couleur de la case sur laquelle elle se situe
 - elle tourne de 90° vers la droite (gauche) si elle est sur une case blanche (noire)
 - elle avance d'une case

On garantit que les données fournies n'imposent pas à la fourmi d'avoir à sortir des limites de la grille.



Énoncé

La fourmi de Langton est un automate cellulaire reposant sur des règles très simples :

- On considère une grille de $W \times H$ cases dont chacune peut être noire '#' ou blanche '.'.
- On dispose la fourmi aux coordonnées (x, y) de la grille
- On oriente la fourmi vers l'un des quatre points cardinaux (N : haut, E : droite, S : bas, W : gauche)
- On laisse la fourmi se déplacer pendant T tours de la façon suivante :
 - elle inverse la couleur de la case sur laquelle elle se situe
 - elle tourne de 90° vers la droite (gauche) si elle est sur une case blanche (noire)
 - elle avance d'une case

On garantit que les données fournies n'imposent pas à la fourmi d'avoir à sortir des limites de la grille.

Question

- Donner le diagramme UML de l'application
- Implémenter cette application